Wisconsin Program-Slicing Tool 1.0
Reference Manual

1997

University of Wisconsin
Madison, WI

Wisconsin Program-Slicing Tool 1.0 Reference Manual

The Wisconsin Program-Slicing Tool is partially implemented through the use of the Synthesizer Generator™, a product of GrammaTech, Inc. GrammaTech and the Synthesizer Generator are trade-names of GrammaTech, Inc., One Hopkins Place, Ithaca, NY 14850.

The Wisconsin Program-Slicing Tool makes use of technology protected by the following patent: Reps, T., Horwitz, S., and Binkley, D., Interprocedural slicing of computer programs using dependence graphs. U.S. Patent No. 5,161,216, United States Patent Office, Washington, DC, Nov. 3, 1992.

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants DCR-8552602, DCR-8603356, CCR-8958530, CCR-9100424, and CCR-9625667, by the Defense Advanced Research Projects Agency under ARPA Order No. 6378 and ARPA Order No. 8856 (monitored by the Office of Naval Research under contracts N00014-88-K-0590 and N00014-92-J-1937, respectively), as well as by grants from Cray Research Foundation, DEC, Eastman Kodak, HP, IBM, 3M, and Xerox.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notices affixed thereon. The views and conclusions contained herein are those of the authors, and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

Contents

Chapter 1

Introduction

The Wisconsin Program-Slicing Tool is a prototype system that supports backward and forward slicing—operations that help the user gain an understanding of what a program does and how it works.

At the heart of the system is a package for manipulating *program dependence graphs* (PDGs) [2, 3, 7] and an extension of program dependence graphs, called *system dependence graphs* (SDGs) [6, 9]. System dependence graphs represent patented technology [5], the rights to which are held by the Wisconsin Alumni Foundation.

This document describes the basic command set of the Slicing Tool. An overview of the system's theoretical underpinnings can be found in [4, 6, 9, 1].

The user interface for the Slicing Tool incorporates a language-specific editor created using the *Synthesizer Generator*, a meta-system for creating interactive, language-based program-development systems [11, 12]. As with all editors created with the Synthesizer Generator, the Slicing Tool's editor exhibits characteristics that are specific to the Slicing Tool, while at the same time sharing the generic user interface described in Chapter 5 of *The Synthesizer Generator Reference Manual* [12].

This document primarily describes the commands that are specific to the Slicing Tool—for example, for invoking **slicing** operations—as opposed to commands that are part of the standard user interface of editors created with the Synthesizer Generator.

Chapter 2

Overview

The Wisconsin Program-Slicing Tool can perform forward or backward slices of a given program with respect to a set of program components, illustrate control and/or flow dependences between components, as well as report certain other kind of information about the program (for example, what global variables are modified by a procedure call).

There are two versions of the Slicing Tool: an interactive version, which allows a user to invoke various operations and view the results; and a batch version, which can be used to report statistics about the program, or to build and save a dependence-graph representation of the program that can be used later by either the interactive or batch versions of the Slicing Tool. (Once a program's dependence-graph representation is created, subsequent operations on the program can be carried out much faster.)

Commands are provided to build or use a system dependence graph to perform various actions, as documented in Subsection 2.6.2. The Synthesizer Generator's facility for displaying program components in different display styles is used to report information back to the user of the Slicing Tool. The various commands use different colors and/or different typefaces (depending on how the display styles are defined in the X resources file for the editor) to indicate various collections of program components, for example, all of the components in a slice of a program, etc.

Space and time considerations are discussed in Section 2.3. Various configurations of the Slicing Tool are possible, both in interactive mode and batch mode. The decision as to which configuration to use should be made according to the size of the program that will be operated on, as well as the operations that will be performed on the program.

The Slicing Tool can operate on complete or partial C programs; assumptions made about incomplete programs are discussed in Section 2.5.

Finally, Section 2.6 gives some insight on the organization of the Slicing Tool's implementation.

2.1 A Session with the Editor

The program **slice_syn** provides the user interface for the Slicing Tool's operations. The results of slicing operations are displayed in editor windows with the results highlighted in a contrasting display style.

This section contains an outline of a session with the Slicing Tool. For the general use of editors created with the Synthesizer Generator, see Chapter 5 of *The Synthesizer Generator Reference Manual* [12]. Commands specific to the Slicing Tool are described in this document in Chapter 3, *Slicing Tool Commands.* The Slicing Tool has an online user manual, organized as an HTML document consisting of a collection of "hypertext pages" connected by "hot links".

The Slicing Tool's editor has an associated set of X11 resources to define the *HIGHLIGHT* style and the location of several needed utilities. Before starting a session, the X resources (fonts, styles, help location, etc.) should be properly installed. (For more detailed information, see the Slicing Tool's Installation Instructions.)

To get started, execute the command

slice_syn *file*

where *file* is a C source file. The editor pops up a window that displays *file.* Typically, the session consists of the following steps:

- Step 1. Building the Sdg

  Before any other operation can be done on the program, its dependence-graph representation (i.e., its SDG) needs to be built. In order to do this, select the **build-sdg** command in the *SDG* menu. A dialog box pops up in which one enters (i) a list of C source *files* that constitute the program, (ii) compile-time *flags* to be passed to the C preprocessor, and (iii) *library files.* Let us ignore the library files for now. The source files can be located in any directory. (Specify them either in terms of path names relative to the directory associated with the current window or in terms of full path names. The directory associated with a window is the directory of the file displayed in that window, or the current directory if no file is displayed in that window.) Hitting the return key (or clicking on the button labeled OK) activates the command. (Some warnings and diagnostic messages from the C front end will appear on the standard output.)

  If the **build-sdg** command succeeds, statistics about the system dependence graph are displayed (e.g., the number of procedures, the number of vertices, the time taken by the construction phases, etc.), and the slicing operations are now enabled.

- Step 2. Slicing

You are now ready to perform a slicing operation on the program whose SDG has just been built. In a window that displays one of the program's files, select—by clicking the left mouse button and dragging the mouse— the program components with respect to which you want to slice the program. Then use the *Slice* menu to choose the **slice** (resp. **forward-slice**) command. The current window is modified to highlight the components of the file that are in the slice (resp. forward slice).

- Step 3. After slicing

  After a successful slice operation has been performed, the **slice-browser** command displays the list of files that have components in the slice. Selecting one of these files opens a new window, displays the file, and highlights the components in the slice.

  At this point, it is possible to use the **slice-augment** (resp. **forward-slice-augment**) command to augment the set of program components with respect to which the slice is performed. The current window is updated to highlight components of the file that are in the new slice. Again, to see program components located in other program files, use the **slice-browser** command.

  To perform a new **slice** (resp. **forward-slice**) command, you should first execute the **unslice** command. This command restores all editor windows to their initial states (i.e., without highlighted program components). Then, follow step 2 again.

  If you wish to perform slicing operations on another program, you should first execute the **unslice** command, then follow step 1.

- Step 4. Saving

  Before exiting the editor, or switching to a new program, you can save the system dependence graph of the current program. In order to do so, select the **write-sdg** option in the *SDG* menu and enter, in the dialog box that pops up, the name of the file in which to save the program. This file will contain global information about the program. In addition, for each source file (for example *foo.c*) two files (*foo.pdgs* and *foo.charpos*) will be created in the current directory (see Section 2.4 for more information about these files). At any time later, this SDG can be reinstalled (without repeating step 1) by invoking the **read-sdg** command with this file name as argument.

  If you want to save a dependence representation of just the entry points of the current program, select the **archive-sdg** command. This command creates information that can be used subsequently as *library information* when the program is used as a module of a larger system. The file created by the **archive-sdg** command records the dependences for the operations of the module's interface, thereby

allowing the larger system to be sliced more efficiently when the user is not interested in viewing the results of slicing within the module. No other files are created. An archive file records an abbreviated version of the SDG, and does not have the same format as the one produced by the **write-sdg** command. An archive file cannot, therefore, be used as an argument to the **read-sdg** command; it can only be used as one of the library files in the **build-sdg** command.[1]

- Step 5. Exiting

  Choose the exit option from the *File* menu to end the session.

2.2 Batch Mode

A batch version of the Slicing Tool can be used to build a system dependence graph for a given program and to perform a limited repertoire of slicing operations. In addition, depending on command-line arguments supplied to it, the batch version of the Slicing Tool can perform the following functions:

- output statistics about the SDG
- report global information about the program (the call-graph, the sets of global variables used or modified by a procedure, etc.)
- write the SDG to a file (either as an SDG or as an archive file). This file can be accessed later by the batch Slicing Tool **sdg**, or by the **slice_syn** editor.

The program named **sdg** is the batch version of **slice_syn**; **sdg** does not include the editor for viewing the results of slicing. (However, **slice_syn** can be invoked later to manipulate the artifacts created with **sdg**.)

See the manual page for **sdg** for a complete description of this utility and its command-line arguments.

2.3 Configurations and Performance

The user of the Slicing Tool — in either its interactive or its batch incarnation — can choose among several possible configurations. The decision as to which one to use should be made according to the size of the program that will be operated on, as well as the operations that will be performed on the program.

The different configurations are distinguished by the following features:

[1] One library of general utility has been supplied with the Slicing Tool: **$WPSTLOC/etc/sdg_library** (where WPSTLOC is the environment variable that specifies the root location of where the Slicing Tool is installed). This library contains an archive file with the dependences for some of the C library functions. (WARNING: this archive file is by no means complete, and some of the dependences for some functions are specific to the Solaris operating system.)

- The representation of dependence graphs in a compressed format: In each PDG, each strongly connected component is condensed to a single node, and certain internal information that distinguishes between various types of edges is discarded.
- The use of external storage for storing dependence graphs: Dependence graphs are written to intermediate files, and read in on demand to compute summary edges and slices.

Thus, for both the interactive and batch versions of the Slicing Tool, there are three possible configurations:

- the *regular* configuration: No compression is performed, and the entire SDG is kept in main memory. The interactive version is named **slice_syn**, and the batch version is named **sdg**.
- the *compressed* configuration: The entire SDG is kept in main memory, but each PDG is compressed. The interactive version is named **cprs_slice_syn**, and the batch version is named **cprs_sdg**.
- the *io* configuration: In addition to the use of compression (as in the compressed configuration), the operations for building and slicing SDGs make use of secondary storage. The interactive version is named **io_slice_syn**, and the batch version is named **io_sdg**.

With all three configurations, the Slicing Tool's user interface is essentially the same, and slicing operations give the same results no matter which version is used. (Chopping operations are currently not supported in the *io* configuration.) The different configurations differ in terms of how large a program they can handle and how fast the various operations are performed. The best time performance is obtained with the *compressed* configuration (see Table 2.1). Due to input/output operations, the *io* configuration is slower but does not require as much main memory as the other two configurations.

Tables 2.1-2.5 show the performance of the Slicing Tool on several example programs. These programs were chosen from various GNU utilities, Spec95 benchmarks, and other benchmark programs. The SDGs for these programs were built with the library **$WPSTLOC/etc/sdg_library**, which simulates the dependences in the most common C library functions. The experiments were run on a SUN SPARCstation 20 model S20FX1-71 equipped with 256 MB of RAM and configured to use 600 MB of virtual memory, running SunOS 5.5.1. [2]

Table 2.1 reports the time in seconds needed to build the SDG and to perform a slice. The times reported correspond to the *regular*, *compressed*,

[2]The measurements given in Tables 2.1-2.5 were made with an alpha version of the Slicing Tool. The performance of the distributed version — the Wisconsin Program-Slicing Tool 1.0 — should be similar.

program	*front end*	*SDG*		*summary comp.*		*slicing*		
		vertices	*time to build* all PDGs	*edges*	*time*	*vertices*	*%*	*time*
cat	3.80	929	5.81	329	0.15	422	45	0.01
			6.13		0.11			0.01
			6.27		0.37			0.29
sort	13.24	4294	226.00	2279	0.89	2112	49	0.06
			237.30		0.61			0.06
			235.81		2.01			2.09
compress	14.40	2270	5.28	2747	0.66	528	23	0.02
			4.78		0.65			0.02
			5.48		1.30			0.69
gcc.cpp	25.41	15070	88.94	51043	15.15	10696	71	0.56
			52.59		9.26			0.36
			78.95		17.96			9.14
li	108.14	178826	579.99	2432359	566.54	145478	81	101.18
			132.90		571.42			89.42
			140.28		505.48			223.27
m88ksim	380.99	63818	476.88	653384	193.05	35846	56	4.98
			243.49		157.98			4.12
			276.93		197.98			80.96
go	164.24	101422	577.51	739566	212.91	65529	64	19.26
			186.25		217.21			18.73
			231.75		213.16			62.83
espresso	126.39	86175	687.44	799449	182.57	52053	60	10.38
			289.36		177.29			6.91
			365.46		197.50			88.06
ijpeg	266.04	51175	134.04	48852	8.02	27562	54	3.37
			114.40		7.87			2.80
			130.25		23.63			26.24

Table 2.1: Time (in seconds) for building and slicing SDGs. In each block of columns three, four, and five, three time figures are given. From top to bottom, these refer to the times used by the *regular*, *compressed*, and *io* versions of the Slicing Tool. The figures listed under % in column eight indicate the percentage of vertices in a slice, averaged over all slices with respect to calls to library functions and calls to undefined functions.

program	*Source*		*SDG*				
	lines	*kbytes*	*PDGs*	*vertices*	*edges*	*call sites*	*globals*
cat	732	17	11	929	5266	61	16
sort	1844	44	39	4294	44653	245	24
compress	1934	73	31	2270	8931	69	36
gcc.cpp	4079	88	85	15070	149807	477	79
li	7597	174	361	178826	2904865	1435	82
m88ksim	19092	463	357	63818	899989	1644	531
go	29629	660	384	101422	1099995	2095	283
espresso	22050	723	372	86175	1259764	2754	117
ijpeg	28177	915	510	51175	257009	1820	79

Table 2.2: Program statistics.

program	*vertices*	*edges*	*call sites*	*parameters*
cat	402	3170	33	20
sort	1029	24037	76	27
compress	323	1713	12	37
gcc.cpp	1658	27689	83	80
li	20133	404418	190	84
m88ksim	4738	169178	97	531
go	2361	74788	48	284
espresso	8129	207214	170	94
ijpeg	2760	17319	186	76

Table 2.3: Maxima per PDG.

and *io* configurations, respectively on the first line, the second line, and the third line of each block in columns four, six, and eight. The time to build the SDG is broken in two parts: the time to build all PDGs, and the time to compute summary edges. The time and number of vertices reported for a slice are the average over all slices with respect to calls to library functions and calls to undefined functions.

Table 2.2 reports statistics about the programs: the number of lines and the size, in kilobytes, of the source code; the number of procedures (PDGs); the number of vertices and edges; the number of call sites; and the number of global variables (local static variables are considered as global variables). Table 2.3 reports the maximum number of vertices, the maximum number of edges, the maximum number of call sites, and the maximum number of parameters over all procedures of each benchmark program. (Global variables that could be used or modified by the procedure, either directly or transitively, are considered as extra parameters.)

Table 2.4 gives an indication of the virtual memory used by the Slicing Tool to build the SDG. The numbers reported in the third column corre-

program	*source files (kbytes)*	*virtual memory (Mbytes)*	
cat	17	regular	6.4
		compressed	6.4
		io	6.1
sort	44	regular	12
		compressed	12
		io	9.6
compress	73	regular	7.5
		compressed	7.5
		io	6.4
gcc.cpp	88	regular	22
		compressed	20
		io	12
li	174	regular	240
		compressed	250
		io	231
m88ksim	463	regular	85
		compressed	87
		io	39
go	660	regular	102
		compressed	107
		io	27
espresso	723	regular	131
		compressed	111
		io	60
ijpeg	915	regular	50
		compressed	51
		io	15

Table 2.4: Use of virtual memory for the example programs.

program	**.c**	**.cfgs**	**.cpp**	**.cppmap**	**.pdgs**	**.charpos**	*sdg file*
cat	18	141	20	3	37	117	2
sort	44	992	48	10	260	517	6.6
compress	73	332	61	7	79	308	8
gcc.cpp	88	1151	88	1	999	1553	106
li	174	1970	577	85	21749	1991	1902
m88ksim	463	4104	2420	141	6194	3954	1527
go	660	5555	850	279	7792	6899	1381
espresso	723	4813	723	1	8393	5620	822
ijpeg	915	4575	1870	202	2006	6754	93

Table 2.5: Sizes of program files and intermediate files (kbytes).

spond to the *regular*, *compressed*, and *io* configurations, respectively on the first line, the second line, and the third line of each block. Table 2.5 reports the size in kilobytes of the source files and the files created by the Slicing Tool.

The time and space numbers should only be used as a potential indication of how the Slicing Tool will perform on comparably sized programs. Besides the number of lines of source text, performance depends on a variety of other factors, including the connectivity of the call graph and the numbers of global variables, call sites, indirect calls, dereferenced pointer variables, etc.

2.4 Files

This section describes the files written at different points during the execution of the Slicing Tool. When not otherwise specified, these files are created in the current directory (i.e., the directory in which the Slicing Tool is invoked), and are created by all three configurations.

Each source file, for example *foo.c*, has the following associated files:

- *foo.cfgs*, which contains the control flow graphs for the functions defined in *foo.c*
- *foo.cpp*, which contains the result of the C preprocessor applied to *foo.c*
- *foo.cppmap*, which contains the correspondence between the C source file and the preprocessed file

In the *io* configuration, or after executing a **write-sdg** command, each source file has two additional associated files:

- *foo.pdgs*, which contains the program dependence graphs for the procedures in *foo.c*
- *foo.charpos*, which contains the mapping between program components and the position of the corresponding text in the source file

The **write-sdg** and **archive-sdg** commands write to the file whose name is passed as an argument to the command. This file can reside anywhere in the file system and will be overwritten if it already exists.

Table 2.5 reports the sizes of the files built by the Slicing Tool for the benchmark programs.

2.5 Assumptions and Limitations

Source files are assumed to contain syntactically correct C code. To allow different parts of system dependence graphs to be built during different

sessions (in a manner similar to separate compilation), no main procedure is required.

It is an error to have more than one non-static function with a given name. Functions not found in the program or libraries are handled as follows: We make the optimistic (non-conservative) assumption that they do not use or modify global variables and do not dereference pointer variables; we then make the conservative assumption that at all calls to such functions all possible dependences exist, and thus insert all possible summary edges (i.e., edges between *parameter-in* and *parameter-out* vertices [6, 9]) into the SDG.

Pointers are handled as follows:

- Every indirect call (call via a pointer) is considered to be a possible call to every function used as an *rvalue* somewhere in the program.
- Every pointer dereference is considered to be a possible access of every piece of heap-allocated storage, as well as a possible access of every variable to which the address-of operator (&) is applied somewhere in the program.

(We are currently working on extending the Slicing Tool with improved pointer-analysis algorithms, and plan to make these improvements available in a future release.)

In the current version of the Slicing Tool, a few limitations on programs are imposed by the C front end. Some of the features that it does not currently handle are

- line continuations
- the use of an identifier that is a type name as the name of a field in a struct, for example:

  ```
  typedef int foo; struct { int foo; };
  ```

- an initialization of a global variable with a *conditional* value, such as:

  ```
  int x = sizeof(int) == 4 ? 0 : 1;
  ```

In addition, the following features can cause erroneous slices to be created:

- variable-length parameter lists
- *setjmp/longjmp*
- signals
- system calls such as *system*, *exec*, *abort*, etc.

2.6 Organization of the Implementation

The Slicing Tool consists of the following modules:

- a C front end to produce intermediate files
- an interface to read intermediate files and build control flow graphs
- a system-dependence-graph module that implements operations on system dependence graphs
- an editor with commands for user interaction
- a version of the Slicing Tool that works in batch mode

Note that only the C front end is language dependent. In principle, it is possible to create a similar Slicing Tool for another language by using an appropriate front end to replace the C front end that is provided in the distribution.

2.6.1 C Front End

The C front end itself consists of two components:

- A customized C preprocessor, named **wpst_cpp**, which processes files and establishes a concordance between the original text and the preprocessed one. The **cpp-display** command illustrates this mapping.
- A tool, named **process**, that scans the program files, one at a time, to produce intermediate files. These files contain two kinds of information:
  - the concordance between source text and program components (i.e., vertices in the system dependence graph), which allows interaction between the user and the slicing tool, for example to define the slice criteria and to display the result of a slice.
  - the information needed to build control-flow graphs and perform data-flow analysis.

Since the C front end operates on a single file, there is essentially no limitation on the size of programs that it can process (as long as each individual file is of reasonable size).

2.6.2 System Dependence Graph

A first pass through the intermediate files produced by the front end builds the *call graph* and gathers information about global variables and indirect calls of the program. A second pass builds a *control flow graph* for each procedure.

The program's *procedure dependence graphs* are constructed from their corresponding control flow graphs. The system dependence graph proper is then constructed by linking together all of the program's procedure dependence graphs with call-graph information and introducing some additional information that summarizes transitive dependences due to procedure calls [6, 9].

The system dependence graph of a given program can be written to a file for later use.

Chapter 3

Slicing Tool Commands

Commands under the *SDG* menu allow the user to build a *system dependence graph* (SDG) for a given program. If the command is successful, slicing, chopping, and other operations on system dependence graphs become valid commands and can be performed on this SDG, which will be referred to below as the *current SDG*.

3.1 System Dependence Graph

This section describes various commands to build the *system dependence graph* for a given program. There are various ways to build an SDG depending on whether the C front end has already processed the program or not, and whether the Slicing Tool has already been used to create *libraries* (i.e, archived abbreviated SDGs) for parts of the program. A program consists of a set of C source *files*, compile-time *flags*, and *libraries*.

The **build-cfgs** command invokes the C front end to produce intermediate data files containing information about control flow graphs. The **build-sdg-from-cfgs** command carries out the actual construction of the SDG once the intermediate files have been produced. **Build-sdg** performs both operations. These three commands are only valid if one of the program files is displayed in the current editing buffer. The intermediate files will be created in the current directory (i.e., the directory in which the Slicing Tool has been invoked). Each program file has several associated files whose names are compounds of the base-name of the program file and the suffixes **.cfgs**, **.cppmap**, and **.charpos**. Once created, these files are never removed by the Slicing Tool (since they can be reused by the **build-sdg-from-cfgs** command). The user must remove them manually (from outside the Slicing Tool) if that is desired.

The **write-sdg** and **read-sdg** commands are complementary. They allow the user to write an already-built SDG to a file, and read it back into memory. The **archive-sdg** command writes an already-built SDG to a file,

in an abbreviated, or archived, format. Such a file contains only summary information and can only be used as a *library* parameter in the **build-sdg** and **build-sdg-from-cfgs** commands.

In dialog boxes that prompt for files, the answer can be entered either as a list of files, with full or relative pathnames, or by typing the name of a file preceded by the special symbol @. In the latter case, the list of files is read from the given file.

Upon successful completion, **build-sdg**, **build-sdg-from-cfgs**, and **read-sdg** set the current SDG for slicing, chopping, and other SDG operations.

The **cpp-display** command highlights the correspondence between the C source file displayed in the current buffer and the C-preprocessed one, which is displayed in a new window.

build-sdg *files flags libraries*

Build a system dependence graph for the specified program. *Files* is a list of C source files, *flags* consists of the compile-time flags to be passed to the C preprocessor, and *libraries* consists of the list of files containing archived SDGs, (i.e., previously built SDGs written to files with the **archive-sdg** command). This command invokes the C preprocessor with *files* and *flags*; then invokes the C front end to write control flow graphs to intermediate data files; and finally builds the SDG.

This command is equivalent to **build-cfgs**, followed by **build-sdg-from-cfgs**.

The command returns an error if the current buffer does not display one of the files listed in *files*, if the C front end detects an error, or if building the SDG fails.

build-cfgs *files flags*

Invoke the C preprocessor on *files* with compile-time *flags*, and call the C front end to write control flow graphs to intermediate files. The command returns an error if the current buffer does not display one of the files listed in *files*, or if the C front end detects an error.

build-sdg-from-cfgs *files libraries*

If intermediate files have already been built for the files listed in *files*, use them to perform the SDG construction, together with *libraries*, a list of files containing archived SDGs (i.e., previously built SDGs written to files with the **archive-sdg** command).

The command returns an error if the current buffer does not display one of the files listed in *files*, or if building the SDG fails.

read-sdg *file*

Read in an SDG from *file*. *File* must have been created by the **write-sdg** command.

write-sdg *file*

Write the current SDG to *file*. The SDG can be read back in by invoking **read-sdg** *file*.

archive-sdg *files flags libraries archive-file*

Build an SDG for the program specified by *files*, *flags*, and *libraries*; and write the SDG to *archive-file* in archived format. *Files* is a list of C source files, *flags* consists of the compile-time flags to be passed to the C preprocessor, and *libraries* consists of the list of files containing archived SDGs, (i.e., previously built SDGs written to files with the **archive-sdg** command). Archived-file format differs from the one produced by **write-sdg** in that only summary information for the SDG's entry point is written out. No SDG can be read back in from *archive-file*. *Archive-file* can only be used as part of the *library* parameter to the **build-sdg** and **build-sdg-from-cfgs** commands.

The command returns an error if the current buffer does not display one of the files listed in *files*, if the C front end detects an error, or if building the SDG fails.

cpp-display *flags*

If the current buffer contains a C file, open a new window containing the file after the C preprocessor (**wpst_cpp**) has been invoked, and highlight the correspondence between the C file and the preprocessed one. *Flags* are the compile-time flags to be used by the C preprocessor.

cpp-clean

After a **cpp-display** command has been executed in the current buffer, remove highlighting from the display of the current buffer and the buffer containing the preprocessed file.

3.2 Program Slicing

The Slicing Tool includes commands to display the components of program slices [13, 8, 6, 9]. A special display style HIGHLIGHT (with a distinctive color and/or type-face) is used to distinguish the components of a slice from other components of the program. The HIGHLIGHT style is specified in an X resources file associated with the Slicing Tool's editor. (See the Slicing Tool's Installation Instructions for information about how to load the X resources that are supplied with the Slicing Tool. See Section 9.6 of *The Synthesizer Generator Reference Manual* for information about defining display styles for editors created with the Synthesizer Generator [12].)

All commands in this section operate on a program for which a *system dependence graph* (SDG) has been successfully built, and, except for **unslice** and **slice-browser**, when the current buffer displays one of the program files of the SDG being sliced.

To perform a slicing operation, first change the current textual selection to the program point or points with respect to which the slicing operation is to be performed; then invoke the appropriate slicing command. The Slicing Tool maintains a *slice set* — the set of components with respect to which the slice is taken. Initially, the *slice set* is empty. Each slicing operation

first changes the *slice set*; the display is then updated so that program components in the backward or forward slice with respect to the *slice set* are displayed in the HIGHLIGHT display style. The following commands are included in the *Slice* menu:

slice
: Set the *slice set* to the collection of program components contained within the current textual selection. Change the display to indicate the components of the backward slice of the program with respect to the *slice set*. To view elements of the slice that reside in other program files, use the **slice-browser** command.

slice-augment
: Augment the *slice set* with the collection of program components contained within the current textual selection. Change the display to indicate the components of the backward slice of the program with respect to the augmented *slice set*. To view elements of the slice that reside in other program files, use the **slice-browser** command. (This command acts just like **slice** if no previous **slice** operation has been performed.)

forward-slice
: Set the *slice set* to the collection of program components contained within the current textual selection. Change the display to indicate the components of the forward slice of the program with respect to the augmented *slice set*. To view elements of the slice that reside in other program files, use the **slice-browser** command.

forward-slice-augment
: Augment the *slice set* with the collection of program components contained within the current textual selection. Change the display to indicate the components of the forward slice of the program with respect to the augmented *slice set*. To view elements of the slice that reside in other program files, use the **slice-browser** command. (This command acts just like **forward-slice** if no previous **forward-slice** operation has been performed.)

executable-slice
: Set the *slice set* to the collection of program components contained within the current textual selection. Change the display to indicate the components of the executable backward slice of the program with respect to the *slice set*. An executable slice includes all program components needed for the actual parameters at all call sites to match the formal parameters at the called procedure [6, 1]. To view elements of the slice that reside in other program files, use the **slice-browser** command.

executable-slice-augment
: Augment the *slice set* with the collection of program components contained within the current textual selection. Change the display to indicate the components of the executable backward slice of the program

with respect to the augmented *slice set*. An executable slice includes all program components needed for the actual parameters at all call sites to match the formal parameters at the called procedure [6, 1]. To view elements of the slice that reside in other program files, use the **slice-browser** command. (This command acts just like **executable-slice** if no previous **executable-slice** operation has been performed.)

slice-extend-to-executable-slice

After a **slice** operation or a **slice-augment** operation has been performed, change the display to indicate the components of the executable backward slice of the program with respect to the same *slice set*. An executable slice includes all program components needed for the actual parameters at all call sites to match the formal parameters at the called procedure [6, 1]. To view elements of the slice that reside in other program files, use the **slice-browser** command. This command is only valid after a **slice** operation or a **slice-augment** operation has been performed.

slice-browser

Pop up a dialog box with the list of program files in the current slice. After a selection is made, clicking on *New Window* brings up a new window that displays the selected file and highlights components of the slice. Clicking on *Same Window* displays the selected file in the current window with the components of the slice highlighted.

unslice

Remove all slicing information from the display of all windows.

slice-save-in-video

Write to a file every program file that has elements in the current slice. Program file *foo.c* is written to file *foo.c.video*, which is in *video* format. (A file in *video* format is a source code file that has been augmented with extra characters so that if the UNIX **cat** command is used to display the file on a vt100-compatible terminal or window, the elements of the slice appear in reverse video.)

display-slice-only

This command produces an approximation to a *projection* of a file using backward slicing: If the current window highlights the program components of a backward slice, it brings up a new window that only displays elements of the slice that reside in that file. (Warning: This collection of elements is not guaranteed to be a well-formed program.) This command is valid only after one of the backward slice operations has been performed.

3.3 Program Chopping

The Slicing Tool provides commands for chopping operations, as described in [10]. These commands are currently not available in the *io* configuration.

A chop is a multi-source/multi-target reachability operation (i.e., it consists of the set of elements along paths from a source component to a target component). The chopping commands operate on a program for which a system dependence graph (SDG) has been successfully built, and the current buffer displays one of the program files of the SDG being chopped. The Slicing Tool maintains a *source set* — the set of components to be the source of a chopping operation, and a *target set* — the set of components to be the target of a chopping operation. Initially, the *source set* and the *target set* are empty. The following commands are included in the *Chop* menu:

mark-source
: Augment the *source set* with the collection of program components contained within the current textual selection. The *source set* will be the set of sources for later invocation of **chop** or **truncated-chop**.

clear-source
: Clear the *source set*, the set of sources for later invocation of **chop** or **truncated-chop**.

mark-target
: Augment the *target set* with the collection of program components contained within the current textual selection. The *target set* will be the set of targets for later invocation of **chop** or **truncated-chop**.

clear-target
: Clear the *target set*, the set of targets for later invocation of **chop** or **truncated-chop**.

chop
: Display program components in the chop of the program (see [10]) with respect to *source set*, the components selected by the **mark-source** commands, and *target set*, the components selected by the **mark-target** commands. To view elements of the chop that reside in other program files, use the **chop-browser** command.

truncated-chop
: Display program components in the truncated chop of the program (see [10]) with respect to *source set*, the components selected by the **mark-source** commands, and *target set*, the components selected by the **mark-target** commands. To view elements of the truncated chop that reside in other program files, use the **chop-browser** command.

chop-browser
: Pop up a dialog box with the list of program files in the current chop. After a selection is made, clicking on *New Window* brings up a new window that displays the selected file and highlights components of the chop. Clicking on *Same Window* displays the selected file in the current window with the components of the chop highlighted.

chop-clear
: Remove all special display styles from the display of all windows. This

command is equivalent to the **unslice** command of the *Slice* menu.

3.4 Displaying Graph Edges

The Slicing Tool includes four commands, **data-flow-predecessors**, **all-predecessors**, **data-flow-predecessors**, and **all-predecessors**, for displaying edges of a program's system dependence graph. These commands are currently not available in the *io* configuration. These commands are included in the *Show* menu. As with the slicing operations, the display of edges in a system dependence graph is initiated by first selecting the program component or components whose graph edges are to be displayed, and then invoking the command. A special display style, HIGHLIGHT, is used to highlight the data-flow predecessors/successors (or alternatively, all predecessors/successors) of the selected components. Note that in the *compress* configuration of the Slicing Tool the nature of edges cannot be distinguished. In this case, the **data-flow-predecessors** and **all-predecessors** commands produce the same result (likewise for **data-flow-successors** and **all-successors**).

data-flow-predecessors
: Display the data-flow predecessors in the system dependence graph of the selected program components in style HIGHLIGHT.

all-predecessors
: Display all predecessors in the system dependence graph of the selected components in style HIGHLIGHT.

data-flow-successors
: Display the data-flow successors in the system dependence graph of the selected program components in style HIGHLIGHT.

all-successors
: Display all successors in the system dependence graph of the selected components in style HIGHLIGHT.

clear
: Remove all special display styles from the display of all windows. This command is equivalent to the **unslice** command of the *Slice* menu.

Acknowledgements

Contributors to the implementation were Thomas Reps, Susan Horwitz, Thomas Bricker, Geneviève Rosay, Victor Barger, Samuel Bates, and Marc Shapiro. Others who contributed to the development of algorithms used in the system were Mooly Sagiv, Jan Prins, David Binkley, Thomas Ball, G. Ramalingam, Philip Pfeiffer IV, and Wuu Yang.

Bibliography

[1] D. Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 2(1-4):31–45, December 1993.

[2] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. In *ACM Trans. Program. Lang. Syst.*, pages 319–349, July 1987.

[3] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, July 1989.

[4] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 392–411, Melbourne, Australia, May 1992.

[5] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing of computer programs using dependence graphs. U.S. Patent Number 5,161,216, issued November 3, 1992.

[6] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1), January 1990.

[7] D.J. Kuck, R.H. Kuhn, B. Leasure, D.A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, pages 207–218, 1981.

[8] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, May 1984.

[9] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 11–20, New Orleans, LA, December 1994.

[10] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 41–52, Washington, DC, October 1995.

[11] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors.* Springer-Verlag, New York, NY, 1988.

[12] T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual: Third Edition.* New York, NY, 1988.

[13] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

Index